Structured Performance Analysis for Component Based Systems

N. Salmi, Patrice Moreaux, M. Ioualalen

Structured Performance Analysis for Component Based Systems

N. Salmi*

Faculty of Computer Science and Electronics,
LSI, USTHB, BP 32, El-Alia. Bab-Ezzouar, 16111, Alger, Algérie
Fax: +213 21247607
E-mail: salmi@lsi-usthb.dz
*Corresponding author

P. Moreaux

LISTIC, Université de Savoie. BP 80449, 74944, Annecy le Vieux, France
Fax: +33 450096559
E-mail: patrice.moreaux@univ-savoie.fr

M. Ioualalen

Faculty of Computer Science and Electronics,
LSI, USTHB, BP 32, El-Alia. Bab-Ezzouar, 16111, Alger, Algérie
Fax: +213 21247607
E-mail: ioualalen@lsi-usthb.dz

Abstract: The Component Based System (CBS) paradigm is now largely used to design software systems. In addition, performance and behavioural analysis remains a required step for the design and the construction of efficient systems. This is especially the case of CBS, which involve interconnected components running concurrent processes. This paper proposes a compositional method for modeling and structured performance analysis of CBS. Modeling is based on Stochastic Well-formed Nets (SWN), a high level model of Stochastic Petri nets, widely used for dependability analysis of concurrent systems. Starting from the definition of the system given in a suitable Architecture Description Language, and from the definition of the elementary components, we build an SWN of the global system together with a set of SWNs modeling the components of the CBS and their connections. From these models, we derive performances of the system thanks to a structured analysis induced by the structure of the CBS. We describe the application of our method through an example designed in the framework of the CORBA Component Model.

Keywords: Component Based System; CBS; Component; SWN; performances; composition; structured method; synchronous interconnection; asynchronous interconnection; method invocation; event-based interaction

Reference to this paper should be made as follows: Salmi, N., Moreaux, P. and Ioualalen, M. (xxxx) Structured performance analysis for component-based systems, Int. J. Critical Computer-Based Systems, Vol. X, No. Y, pp.000000.

Biographical notes: N. Salmi received her PhD in Computer Science in 2008 from the University of Savoie, Annecy, France. Currently, she is a lecturer in the Department of Computer Science, Faculty of Electronics and Computer Science, USTHB University of Algiers, Algeria. She is also an associate researcher at the LISTIC laboratory of Polytech'Savoie, Annecy, France. Her research interests include formal methods for modeling and performance analysis of systems, especially component based systems.

Patrice Moreaux received his PhD in Computer Science in 1996 from Paris-Dauphine University, Paris, France. Currently, he is a professor in Polytech'Savoie, Annecy, France. His research interests includes modelling, design, analysis of distributed systems, component based software systems, service oriented systems and modelling and analysis of Discrete Event Systems (with time).

Malika Ioualalen received her PhD in Computer Science in 1998 at the USTHB University of Algiers, Algeria. Currently, she is a professor in the Department of Computer Science, Faculty of Electronics and Computer Science, at the USTHB University. Her research interests includes performance analysis and modeling of complex systems and quality of service in many fields (mobile networks, multiagent systems, ...).

This paper is a revised and expanded version of a paper entitled [Structured analysis of Component Based Systems: an EJB/CORBA middleware application] presented at VECoS'2007, Algiers, Algeria, May 5-6, 2007.

Introduction

Recent software systems are more and more being designed as a set of components assembled together and interacting to achieve a common goal [4]. This compositional approach has captivated many industrial developers as it allows higher maintainability, reuse, easier upgrade and dynamic reconfiguration. A system is therefore seen as an assembly of components. Such architectures are known as Component Based Systems (CBS). Since the mid'70, several component models have been proposed in the literature [18, 31, 5, 29], and some of them are used in industry.

In this context, a software *component* can be a source code unit (consumed at development time and architectural design time), or a unit of deployment (machine executable). It can also be a unit of versioning and replacement [32]. It is defined as a unit of composition with contractually specified *interfaces* and explicit context dependencies.

Although there is no unique definition of what is a component, authors consider that the definition of a component is made of a behaviour (implementing functionalities) and one or several interfaces. Interfaces are used to assemble components, depending on their interaction, specifying required or offered services. Components can be assembled or composed to form a software application or system, according to their specified *interactions*.

Many design frameworks for CBS exist, each one being based on a component model defining more or less precisely the semantics of the components and of the assembly. To allow definition of a CBS, several *component models* have been introduced, even for academic or industrial purposes. Among the large number of component models proposed in the literature, either industrial or academic, we may quote EJB, CCM, COM+/.net, Fractal, JMX, PECOS, Koala, IEC61499,... [20, 16, 6]. For most of these models, an Architecture Description Language (ADL) [15] allows us to describe an assembly of components. From this description, a set of accompanying tools generate, at least, the templates of the application code and perform some verifications such as data type compatibilities.

The component based approach is a smart approach providing significant benefits. However, designers of CBS should have some assurance that the assembly they define meets performances expected by users (for instance avoiding contention and bottlenecks). Moreover, if large architectures of CBS are involved, such properties are difficult to derive. So, it is necessary to develop methodologies and tools allowing qualitative and quantitative analysis of CBS, to support designers in their activities.

In this area, prediction methods for performance and behavioural qualitative analysis of CBS are still scarce: [1, 2] used Labelled Transition Systems (LTS) and model checking in order to prove temporal logic properties of a Fractal CBS; [10] used hierarchical coloured Petri nets and temporal logic for the specification and verification of software components designed for embedded systems. [23] translates the description of a system given in the ADL AADL into a Generalized Stochastic Petri Net (GSPN) for dependability measures purpose. These attempts addressed mainly qualitative or deterministic time based analysis, whereas performance analysis is usually carried out through measures and testing techniques on existing systems. . It is well known that limiting performance analysis

to existing systems impacts the quality of the software product [33]. In fact, some predictive performance modeling was also proposed as in [30] . This approach translates architecture designs, mostly given in the Unified Modeling Language (UML), into models adequate for performance analysis. Adapted formalisms used for thes analysis are Layered Queuing Networks (LQN) [22], Stochastic Petri Nets (SPN) and Stochastic Process Algebras (SPA). Following this method, [3], [11] proposed to start from a design model and build a performance analysis model based on LQN. However, we claim that UML and queueing networks are not sufficiently expressive to model complex systems, where important characteristics are synchronization, resource contention and conflicts. Moreover, the method should be able to handle large architectures.

For these purposes, we propose in this paper a structured approach for performance analysis of CBS, trying as much as possible to reduce the complexity of analysis and allowing us to analyse large architectures. We first introduced our approach in a previous work [25], with an example of performance analysis of an Enterprise JavaBeans /Common Object Request Broker infrastructure.

The approach, summarized in figure 1 (this figure only appear in colour in the online version of the paper, not the printed version), starts from the architecture description of the CBS, provided in an ADL, models components, derives the CBS global model and finally applies a structured compositional method to derive performance indices. The models can be saved in a repository for reuse in analysis of other CBS. Components are modeled with Stochastic Well-formed Net (SWN) [7], a special class of Stochastic coloured Petri Nets, widely used for performance analysis of complex systems showing symmetrical behaviour of active entities. We first translate component interfaces and interactions in the SWN context. We model these interactions and we show how to build the global SWN of the CBS. Finally, we apply a structured method, derived from our previous works [8, 9], allowing us to compute performance indices in an efficient way, through exploitation of the compositional architecture. Computations are based on a combined aggregation/tensorial representation of the underlying Markov chain of the global SWN, which reduces the complexity of analysis, with time and memory savings. We extend this method to the analysis of CBS with synchronous *and* asynchronous interactions (compositions) of SWN corresponding to components, since the previous method was defined to the analysis of either a synchronous decomposition or asynchronous decomposition of a global SWN. In previous works [24, 27], we have studied modeling and performance evaluation of the Julia implementation of FRACTAL based CBS, which provides synchronous interactions between FRACTAL components. The present paper extends this work to more general CBS with synchronous and asynchronous event-based interactions between components.

Among several component models, we have chosen to illustrate our approach with the Corba Component Model (CCM) [20]. CCM is indeed a language-neutral model, taking into account the two classical interaction modes between components: request/response and event-based.

The structure of the paper is as follows. We present in section 1 the main features of a CBS, illustrated with the CCM model. We also describe a CCM application, used as an example along the paper. We follow by giving an overview of our approach in section 2 and the details of the modeling steps of a CBS in section 3. We explain

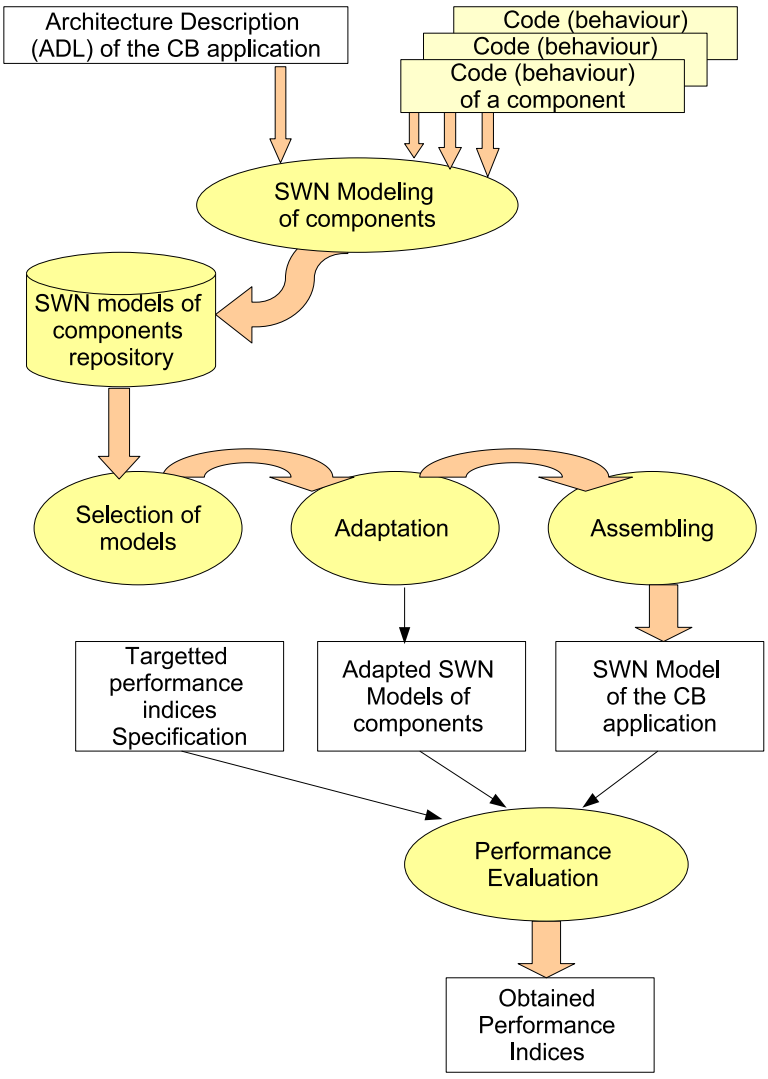


Figure 1: Principle of compositional analysis approach of CBS

in section 4 our analysis method of a CBS, together with application of the approach to our example. We conclude and propose future work in section 5.

1 Component Based Systems

1.1 Concepts

A software *component* is defined as a unit of composition, provided with contractually specified *interfaces* and explicit context *dependencies* [32]. An interface is an access point to the component, which defines provided or required services. In addition, types, constraints and semantics are defined by the *component model* in order to describe the expected behaviour at runtime.

Interfaces of a component allow it to connect to other components. Consequently, we build a CBS by connecting the interfaces of components. These connections are defined through interactions between components. Generally, two main styles of interactions are defined in component models: synchronous interactions provided by service invocation (such as RPC or RMI communication) and asynchronous interactions given through notification of events (asynchronous messages). Service invocations take place between a *client* interface requesting a service and a *server* interface providing the service. Besides, event communications are defined between one or more *event source* interfaces generating events and one or several *event sink* interfaces receiving event notifications. The reception of a notification causes the acknowledgement of the reception and the execution of a specified reaction called the *handler* of the event. Some event services use *event channels* for mediating event messages between sources and sinks. An event channel is an entity responsible for registering subscriptions of a specific type of events, receiving events, filtering events according to specific modes, and routing them to the interested sinks.

A component can contain itself a finite number of other interacting components, called *sub-components*, allowing the components to be nested into several levels. Such a component is called *composite*. Components of the lowest level are called *primitive*. Assembling two components may require an adaptation of associated interfaces, whenever these interfaces cannot directly communicate for example. In this case, the adaptation is done through an extra entity, called *connector*, modeling the interaction protocol between the two components.

For each component model, a corresponding *Architecture Description Language* (ADL) allows one to describe an assembly of components constituting an application. From such a description, a set of tools are used to compile and generate the application code, while checking syntactical and even some semantical properties.

1.2 CBS illustration : CCM based Systems

1.2.1 The CCM model

CCM [20] is a key part of the CORBA 3.0 standard [17], independent from operating systems and programming languages and allowing the deployment of components on a distributed environment.

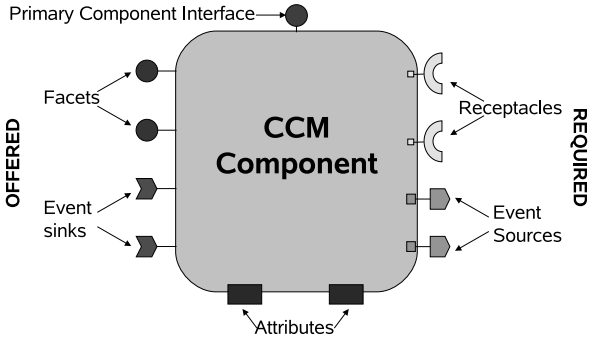


Figure 2: The CORBA component model

A CCM component is an implementation entity, described by a set of *attributes* and a set of interfaces said *ports* (termed interfaces in the sequel). Attributes are named values exposed through accessor and mutator operations. There are two types of interfaces (figure 2), described in CORBA IDL3: *facets* and *receptacles* playing respectively the roles of either client and server interfaces or else event *source* and *sink* ports.

In order to support events, CCM follows the *publish/subscribe event push* model [20], compatible with CORBA notification service [19]. This model defines two roles: *publisher* (event source) and *subscriber* (event sink). Subscribers register their interest for a class of events by *subscribing* to the *publisher* (or the channel) of this class of events. Communication between publishers and subscribers can be brokered through *event channels*. Note that subscribe/unsubscribe components operations are invoked at CBS deployment and (re)configuration time.

Unlike FRACTAL, CCM is a flat component model without hierarchy. Moreover, a CCM component is located inside a *container* which provides it the runtime environment and allows it to access a set of non-functional services such as persistence, event notification, transaction and security. Each container is responsible for initializing instances of the component types it manages, controlling their life cycles, and connecting them to other components and common middleware services, including event publisher/subscriber services and event channels. These services include non-functional properties provided by *controllers* in FRACTAL. A CCM application is built by defining an *assembly*, grouping components and defining metadata describing components.

1.2.2 A CCM application

Along this paper, we exemplify our approach with a stock quoter system, a CCM application managing a stock information database, inspired from a CCM example presented in [28]. It consists of two *StockDistributor* components monitoring a real-time stock database, two *StockBroker* components waiting for stock changes and an *Executor* component processing stock information (figure 3). When the value of a particular stock changes, a StockDistributor pushes a CCM event that contains the stock name, via a CCM event source, to corresponding CCM event sinks

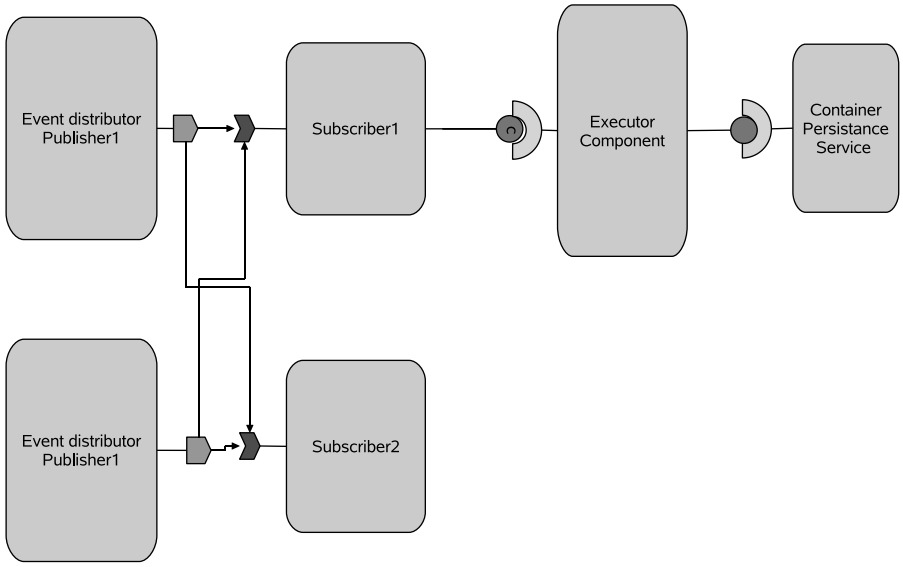


Figure 3: A CCM application

implemented by the StockBroker components. Each StockDistributor is responsible of a set of stocks.

At reception of event notification, one of the StockBrokers invokes a service from the executor component through corresponding receptacle and facet interfaces, whereas the second StockBroker processes locally the event. The executor component processes the StockBroker request, generates data and invokes itself the persistence service offered by the container in order to save data. When no event is generated, the StockBrokers perform continually a local task.

2 Component based analysis approach overview

In this paper, we propose a qualitative and quantitative analysis approach for CBS, concentrating on performance properties. We use for this purpose the Stochastic Well-Formed Net model (SWN) [7], a high level Petri Net model which describes systems in a concise way, and profits from symmetrical behaviours of a system for an efficient analysis.

2.1 Motivations for the SWN model

The choice of the SWN formalism is mainly motivated by three reasons. Firstly, in order to be able to evaluate performance indices related to configurations of systems, such as the number of requests pending in some part of the system, the mean utilization time of some resource, etc., a state based model like Petri nets is suitable. Petri Nets are also well known for being able to model complex systems with concurrency and conflicts, even in the stochastic context, in contrast with Queuing networks or process algebras models for instance. Secondly, although

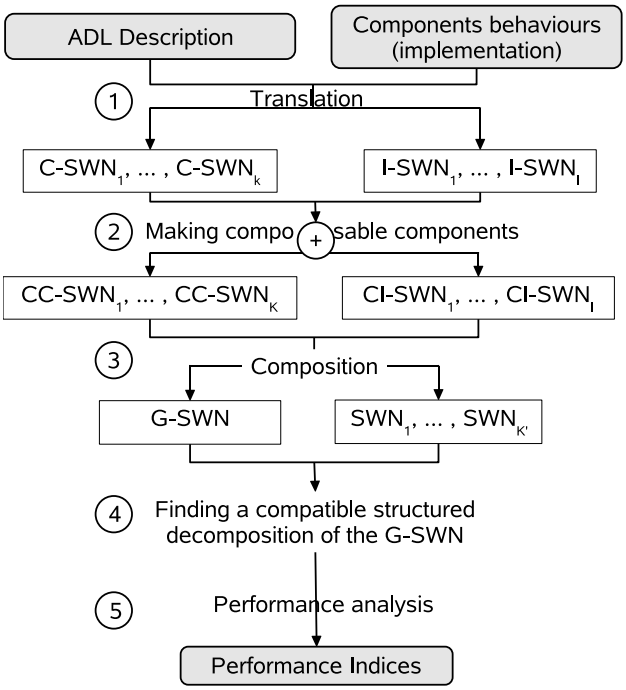


Figure 4: Analysis approach of CBS

Petri Nets are not by themselves a compositional model, interaction between Petri nets representing sub-components may be defined as transition or place "fusion" (merging). Thirdly, if complex primitive components are involved, high level Petri Nets are almost inevitably required. As a high level model, the SWN Petri net model can also take advantage of behavioural symmetries of system's entities if there are such symmetries, by compacting its reachability graph, leading to a *Symbolic Reachability Graph* (SRG). An SRG is composed of *symbolic markings*, where each symbolic marking represents a set of ordinary (coloured) markings having equivalent behaviour. Finally, SWNs are a well studied class of high level stochastic Petri nets and benefit from a consistent set of analysis algorithms and tools [14].

2.2 Analysis steps

Our approach for analyzing a CBS consists of two main phases :

- building a global SWN model for the CBS, seen as a composition of SWN models associated to components and their interconnections, and
- applying a structured analysis method for the computation of performance indices.

These two phases are detailed through figure 4:

1. From the CBS description given using the ADL related to the component model, translation of the component behaviours (source code) and their

interactions in the SWN context. We obtain a set of SWNs called *Component SWNs* (C-SWNs), (for components) and *Interaction SWN* (I-SWNs) (for complex interactions) involving *connectors*. This modeling depends heavily on the component model.

2. Modification of the C-SWNs and I-SWNs in order to be composable in the sense of Petri nets composition (fusion of places or transitions), giving rise to *Composable Component SWNs* (CC-SWNs) and *Composable Interactions SWNs* (CI-SWNs).

3. Composition of the CC-SWNs and CI-SWNs (seen as a unique set $\{SWN_1, \ldots, SWN_K\}$) by merging interfaces elements. We obtain a *global SWN* (G-SWN) modeling the CBS.

4. Finding of the set of SWNs $(\mathcal{N}_k)_{1 \leq k \leq K'}$ representing a possible decomposition of the G-SWN, that fulfill conditions for a structured representation of the SRG and its aggregated generator. These SWNs can be one of the $(\mathrm{SWN}_k)_{1 \leq k \leq K}$ or grouping together a subset of them.

5. Computation of performance indices by applying the structured analysis method [8, 9].

In order to partly automate our approach, we explain each of these steps in the following.

3 SWN modeling for CBS

The first three steps of our approach are detailed in this section. Our aim here is to derive automatically a stochastic model for the CBS, to be used for efficient structured compositional analysis. A set of rules are therefore proposed for automatic translation of CBS descriptions in the SWN framework. We first discuss our modeling choices, then we detail modeling of interfaces of components and modeling of primitive and composite components.

3.1 General modeling choices

3.1.1 modeling stable configurations

Before being available to users, a CB application must be deployed and configured. These steps include, among others: initialization and runtime contexts definition, subscribing of components exposing an event sink for event notifications, etc. When these steps are completed, the application is activated. We say that it is in a *stable configuration*. During its execution, the application *architecture* may evolve through (runtime) reconfigurations, for instance by creating or deleting instances of components, subscribing/unsubscribing of components for a special type of event.

Although the capability of runtime evolution of the architecture is a leading property of several middlewares for CBS, it seems that behavioural analysis of CBSs cannot be efficiently carried out with a single formal model of the system. In fact, any modification of the architecture requires a specific modeling, primarily devoted

to check that, starting from a configuration A, the application will eventually reach a given configuration B. In contrast, analysis of a given configuration (say A or B) addresses both qualitative (reachability, deadlock freeness, etc.), and quantitative (computation of performance indices) aspects of this configuration. Moreover, in the performance evaluation context, switching from A to B probably corresponds to a "short" time period (transient phase, finite horizon analysis), whereas we are interested in performance indices of software systems computed over long periods (steady-state analysis).

In the present work, we do not address the verification of the reconfiguration behaviours of CBSs and we concentrate on "stable" (i.e. fixed) architectures. Hence, we do not model non-functional services pertaining to initialization and reconfiguration steps. However, we can obviously *compare* performances of two configurations A and B, each one studied with our method.

3.1.2 Implementation dependencies

Since we wish to derive performance indices of a CBS, we emphasize that the architecture description of the CBS is not sufficient for performance modeling: *it must be complemented with information from the implementation* of the component model. Indeed, an implementation of a model can differ from another one. This is the case for instance, for the FRACTAL model: the *Julia* implementation uses synchronous method invocation, while *Fractive* uses an asynchronous (late) operation invocation.

3.1.3 Colours

Basic colour classes are used to model data entities and active entities of components. Data entities consist of data flow such as requests, parameters of requests, requested data, event data or even resources. Active entities are execution flows (processes and threads). In our context, we emphasize on modeling entities involved in interfaces and interactions, since entities used inside a component depend only on that component. More precisely, we adopt basic colour classes for modeling methods and their parameters invoked through interfaces, event types, possible resources and execution flows of a component. For this last purpose, we should know if the component is multithreaded or not. It could also be useful to "colour" event related data when messages associated to the events should be taken into account from a performance point of view.

3.1.4 modeling event entities

Event messages can be brokered through an event channel, depending on the specification of the component model. In this case, we discuss here the need for the explicit modeling of the event channel. In fact, when the event channel receives an event notification from the publisher, it generally acknowledges it (as it is done in CORBA [19]) and then it routes the notification to all interested subscribers. The event acknowledgement can be sent to the publisher as soon as the notification is received by the channel, or it can be delayed until the subscribers receive themselves the notification and acknowledge it. In order to obtain a flexible model for our components, it could be interesting to model explicitly the event channel to consider

various cases of event acknowledgement. However, this may affect response times and performances of the whole system. Therefore, in order to take into consideration this influence, we choose to model explicitly the event channel.

3.1.5 Container services

A container acts as a component manager. It offers three kinds of services :

- Operations relating to component lifecycle, such as creating, deleting or modifying a component or a connection, etc.
- Operations relating to service invocations of components, and
- Calls to non-functional services offered by the operating system or a middleware, like transaction manager, security manager, event manager and persistence manager services.

Lifecycle and connections management is related to non stable configurations non covered by this paper, for reasons stated in 3.1.1. However, we model other container services which don't imply modification of the architecture. These services are considered as abstracted components endowed with their proper interfaces, and are modeled with SWN models as for components (see details in section 3.6).

We detail now the modeling phase, beginning with modeling interfaces of components, then components and finally the application.

3.2 Modeling client/server interfaces

A service invocation takes place by specifying the required method and its parameters. So, we translate corresponding interfaces of a component following the mapping rule.

Mapping rule 1: client/server interfaces

• A server interface, identified by a set of colours STh modeling possible server threads, offering a set MP of operations or methods with their parameters, is modeled by representing the beginning of service provided and its ending with two transitions, respectively t_{BPS} and t_{EPS} (figure 5 (right)). t_{BPS} is controlled by two places *ServTH* and *SoMP* modeling respectively server threads and methods with their parameters. Possibly, a third place *ResServ* coloured with a basic class R is used modeling specific resources needed during execution of a service. Whereas, t_{EPS} is controlled by a place *Result* coloured with tuples belonging to $STh \times MP \times R$ modeling the result of request processing.

• A client interface, identified by a set of colours CTh modeling possible request threads of the client component is modeled with two transitions t_{BRS} and t_{ERS} representing the beginning of service request and its ending (figure 5 (left)). t_{BRS} (resp.t_{ERS}) is controlled by a place *PBegReq* (resp. *PEndReq*) coloured with CTh and modeling respectively requesting and released client threads.

The model of a request/response interface depends a priori on the invoked method and its parameters. In our mapping rule, we do not separate them: if such a level of detail is required for performance analysis, a colour class is defined with static subclasses sorting the possible pairs (method, its parameters) into disjoint

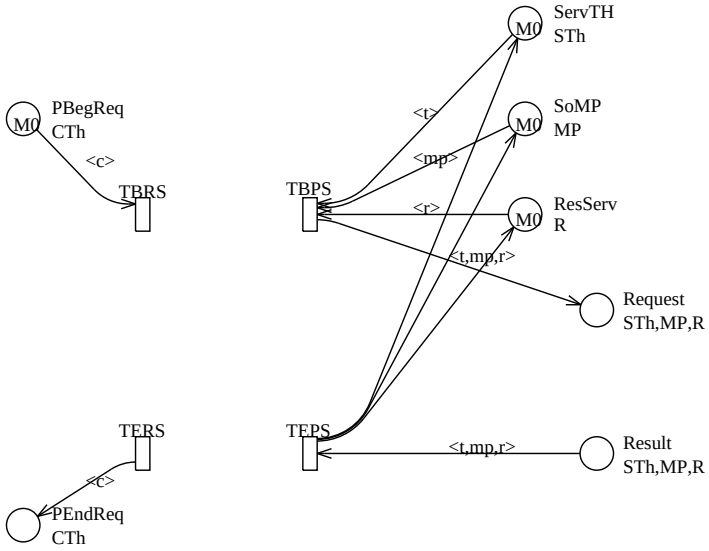


Figure 5: C-SWNs models of client (left) and server (right) interfaces

subsets. Note that if the pair is irrelevant for a given level of detail, we simply omit this colour class.

Let us give the formal definition of client and server interfaces; $t^\bullet$ (resp. ${}^\bullet t$) denotes the set of output (resp. input) places of transition t.

Client interface: A client interface I of a C-SWN is of a couple of transitions (t_{BRS}, t_{ERS}), such that:
• $\mathcal{D}(t_{BRS}) = \mathcal{D}(t_{ERS}) = CT \times MP$ ($\mathcal{D}(x)$ represents the colour domain of the node x);
• $t^\bullet_{BRS} = {}^\bullet t_{ERS} = \emptyset$;
• $\exists! p_{BegReq} \in P$, $\exists! p_{EndReq} \in P$, $p_{BegReq} \neq p_{EndReq}$,
$Pre(p_{BegReq}, t_{BRS}) = Post(p_{EndReq}, t_{ERS}) = Id$.

Server interface: A server interface I of a C-SWN is a tuple $< P_I, T_I >$ such that:
• $P_I = \{ServTH, SoMP, ResServ\}$;
• $T_I = \{t_{BPS}, t_{EPS}\}$ with $\mathcal{D}(t_{BPS}) = \mathcal{D}(t_{EPS}) = ST \times MP \times R$;
• $Pre(ServTH, t_{BPS}) = Post(ServTH, t_{EPS}) = Id$;
• $Pre(SoMP, t_{BPS}) = Post(SoMP, t_{EPS}) = Id$.

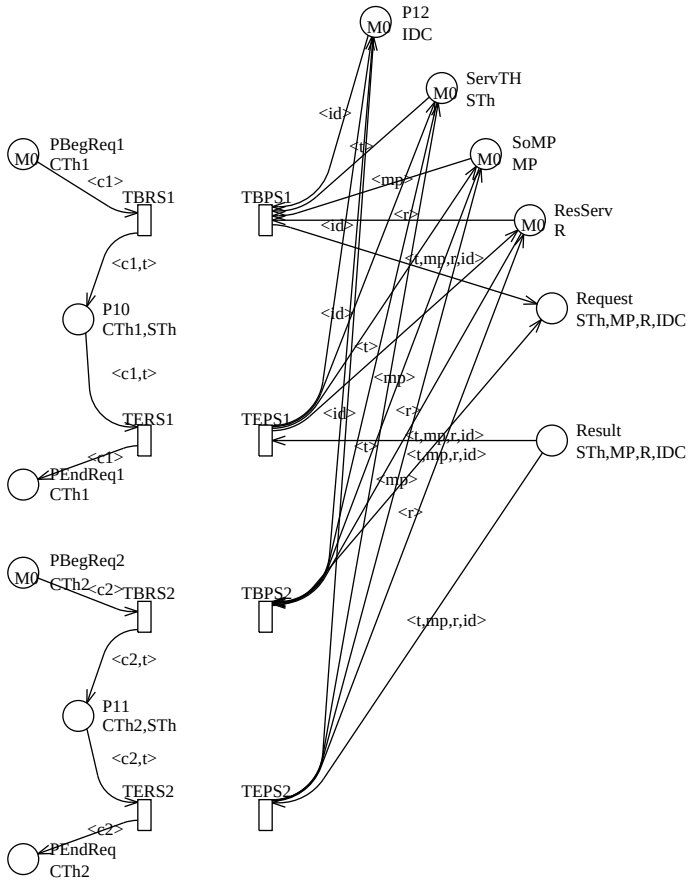


Figure 6: Interfaces of the CC-SWNs models with multiple client interfaces connected to the same server interface

Mapping rule 2 defines the CC-SWN, extending the client interface part of a C-SWN to allow composition of SWN and subsequent structured analysis without modifying the semantics of the component.

Mapping rule 2: CC-SWN for clients The client interface of a C-SWN is modified, leading to a CC-SWN, by adding a place (and associated arcs) as a postcondition of the beginning transition t_{BRS} and as a precondition of the transition t_{ERS}. The colour domain of this place is either $CTh \times STh$ or else $CTh \times STh \times IDC$ when several client components require the same service (see below).

Dealing with multiple clients When a server interface of a component is connected to several client interfaces of other components, the C-SWN of this component must be modified in order to be composable with client interfaces of several components, in the sense of Petri nets composition (fusion of places or

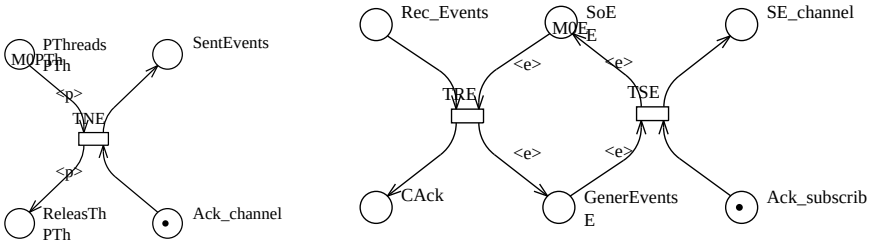


Figure 7: SWN model of an event channel

transitions). This is achieved by applying mapping rules 2 and 3. The resulting CC-SWN keeps the same semantics as the corresponding C-SWNs.

Mapping rule 3: Multiple clients for one server interface The interface part of the CC-SWN of a server interface which have multiple connected clients is modified as follows with respect to the single client case: (i) The transitions t_{BPS} and t_{EPS} of beginning and ending service are duplicated as many times as the number of client interfaces; (ii) An IDC colour class is introduced to distinguish between several components exposing a client interface. The resulting interface is given by figure 6.

3.3 Modeling event-based interfaces

Event based communication is generally realized following a specific pattern. Among the most representative patterns, we find the *publish/subscribe* model, used in the CORBA notification specification. In this specification, a publisher pushes an event of a specified type to subscribers through an event channel (a channel manages a specified type of events). The publisher component continues its work after emitting an event. The event channel receives the notification and acknowledges it to the publisher. Then, it sends the notification to all interested subscribers. When receiving an event, a subscriber acknowledges it and triggers a handler processing the event. This interaction model is translated with mapping rule 4 modeling the publisher, the event channel and the subscriber.

Mapping rule 4: Event interfaces (1)

• A publisher interface of a component, identified by a set PTh of colours modeling possible publisher threads, is modeled with a transition *TNE* representing the notification of events (figure 7, left), with *Pthreads* and *Ack_channel* places as preconditions, and *ReleasTh* and *SentEvents* as postconditions.

• An event channel managing a set E of events of a specified type is modeled with two transitions *TRE* and *TSE*, expressing respectively receiving events from publishers and sending these events to subscribers (figure 7, right). The *TRE* transition is controlled by *Rec_Events* and *SoE* places and has the *CAck* and *GenerEvents* places as postconditions. While, the *TSE* transition is controlled by *GenerEvents* and *Ack_subscrib* places and has as postconditions the *SoE* and *SE_channel* places.

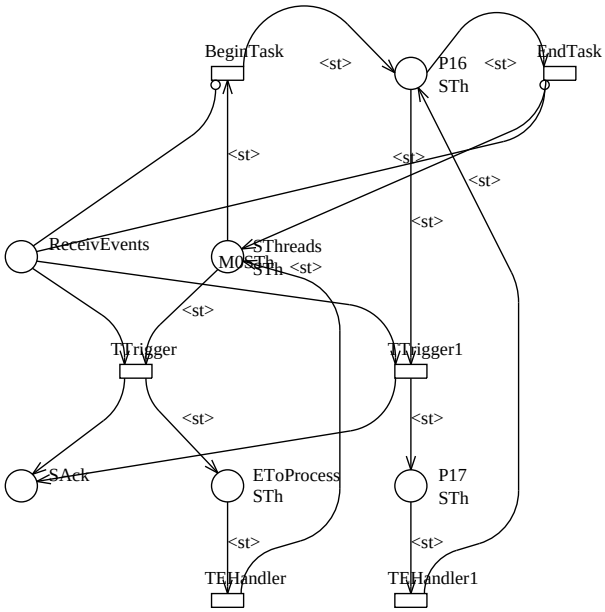


Figure 8: Example of the SWN model of a subscriber

• The SWN of a subscriber interface of a component, identified by a set STh of colours modeling possible subscriber threads (see figure 8 for an example), is made of two parts: (i) a local processing modeled by the *BeginTask* and *EndTask* transitions, controlled by two places *ReceivEvents* and *SThreads* places, and (ii) an event processing part triggered with the *TTrigger* transition modeling the reception of an event. *TTrigger* is also controlled by *ReceivEvents* and *SThreads* and has *SAck* and *EToProcess* as postconditions. The event handler is modeled with the *TEHandler* transition.

In the publisher model, the two preconditions of the *TNE* transition *Pthreads* and *Ack_channel* model respectively the publisher threads and the ready state of the component. Whereas, postconditions, *ReleasTh* and *SentEvents*, model respectively the publisher threads resuming their activities and event notifications. The *Ack_channel* and *SentEvents* places are uncoloured as we model publishers notifying one specific type of events

In the event channel model, the *Rec_Events* and *SoE* places model respectively received notifications and the set of possible events. *Rec_Events* is not coloured. Postconditions of *TE* (*CAck* and *GenerEvents*) model respectively acknowledgement and generated events to be sent to subscribers. *CAck* is also not coloured, and *GenerEvents* has the same domain as *SoE*. The *Ack_subscrib* place models the state of the channel ready to broadcast notifications to subscribers. It is also uncoloured like *Ack_channel* of the publisher model. The *SE_channel* place models events sent to subscribers by the channel.

In the subscriber model, local processing should be interrupted when events are received. We model this interruption with inhibitor arcs preventing firings of local processing transitions (*BeginTask*, *EndTask* and possibly others expressing more local activities)) as soon as an event is received in place *ReceivEvents*. The *SThreads* place is coloured with the STh basic class, while *ReceivEvents* has the same domain as the *SoE* place (neutral or set of event colours when dealing with several types of events). Reception of an event causes sending an acknowledgement in the uncoloured place via firing of the transition *TTrigger* (or *TTrigger1*), *SAck* and execution of an event handler. In this model, processing an event is abstracted into one transition *TEHandler* (or *TEHandler1*). We could replace this transition with a subnet detailing the handler if we are interested in impact of processing details on performances of the system.

Note that triggering the handler should usually be achived in a "short" period of time, with respect to components activities. This should be reflected in firing rates ratios (for instance 1/0.001) of *TTrigger, TTrigger1* transitions, over *BeginTask, EndTask* and *TEHandler, TEHandler1* transitions.

Formally, we define publisher and subscriber interfaces and event channel, as follows.

Publisher interface: A publisher interface I of a C-SWN N consists of a tuple $< P_I, T_I >$ such that:
• $P_I = \{PThreads, SentEvents, ReleasTh, Ack\_channel\}$, with $\mathcal{D}(PThreads) = \mathcal{D}(ReleasTh) = PTh$, $\mathcal{D}(Ack\_channel) = \mathcal{D}(SentEvents) = \epsilon$;
• $M_0(Ack\_channel) = 1$;
• $T_I = \{\text{TNE}\}$ with $\mathcal{D}$(TNE)=PTh;
• Pre(PThreads,TNE)=Post(ReleasTh,TNE) = Id ;
• Post(SentEvents,TNE)=Pre(*Ack_channel*,TNE) =1.
where ϵ denotes the neutral colour.

Subscriber interface: A subscriber interface I of a C-SWN N is a tuple $< P_I, T_I >$ such that:
• $P_I = \{SThreads, ReceivEvents, EToProcess, SAck\}$, with $\mathcal{D}(SThreads) = \mathcal{D}(EToProcess) = STh$, $\mathcal{D}(SAck) = \mathcal{D}(ReceivEvents) = \epsilon$;
• $T_I = \{TTrigger, TEHandler\}$ with $\mathcal{D}(TTrigger) = \mathcal{D}(TEHandler) = STh$;
• Pre(SThreads,TTrigger)=Post(EToProcess,TTrigger)=Id;
• Pre(ReceivEvents,TTrigger)= Post(SAck,TTrigger) =1.

Event channel: An event channel model is a tuple $< P_I, T_I >$ such that:
• $P_I = \{SoE, Rec\_Events, SE\_channel, GenerEvents\}$, with $\mathcal{D}(SoE) = \mathcal{D}(GenerEvents) = E$, $\mathcal{D}(Rec\_Events) = \mathcal{D}(SE\_channel) = \mathcal{D}(CAck) = \mathcal{D}(Ack\_subscrib) = \epsilon$;
• $M_0(Ack\_subscrib) = 1$;
• $T_I = \{TRE, TSE\}$ with $\mathcal{D}(TRE) = \mathcal{D}(TSE) = E$;
• Pre(SoE,TRE)= Post(GenerEvents,TSE)= Pre(GenerEvents,TRE)= Post(SoE,TSE)= Id;

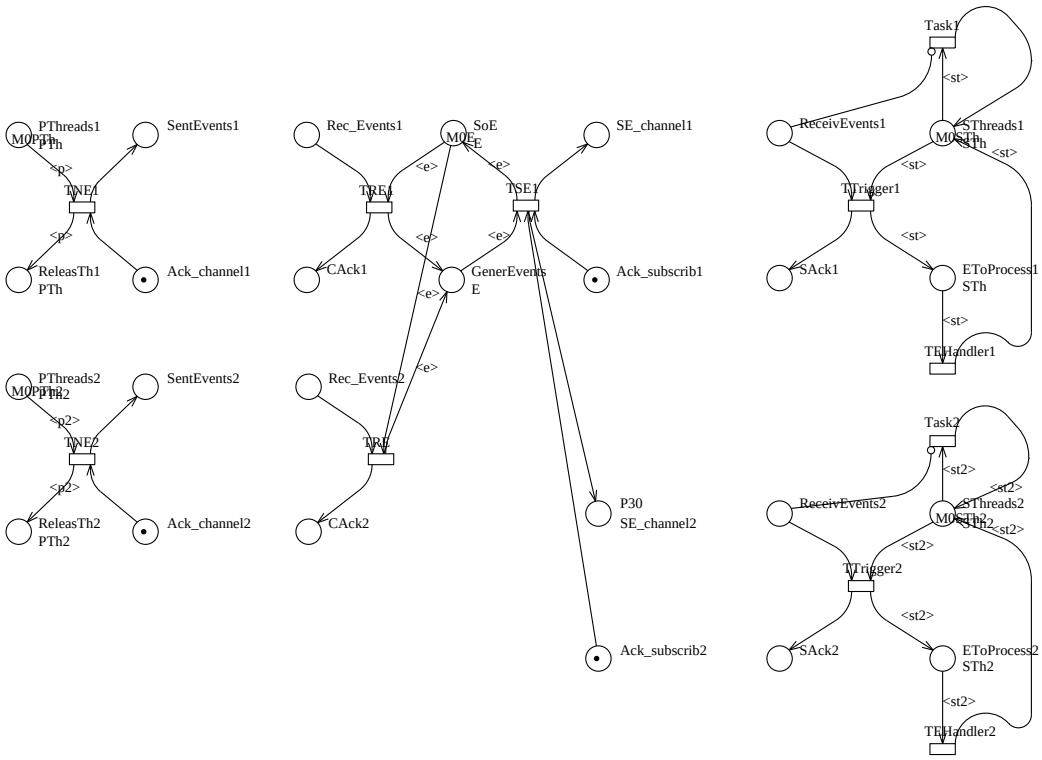


Figure 9: SWNs models of event interfaces with multiple publishers and subscribers

• Pre(*Rec_Events*,TRE)= Post(CAck,TRE)=
Pre(*Ack_subscrib*,TSE)= Post(*SE_channel*, TSE)= 1.

Dealing with multiple publishers and subscribers
A publisher interface of a component can push events to several subscribers. In the same way, a subscriber interface can receive events from various publishers generating the same type of event. This is the role of the event channel to mediate communication between publishers and subscribers. So, in these cases, the C-SWN of the event channel must be modified (see mapping rule 5) in order to be composable at the same time with several publishers and subscribers models (fusion of places or transitions). This modification gives rise to a CC-SWN modeling the event channel. Note that CC-SWNs corresponding to publishers and subscribers components remain the same obtained C-SWNs without any modification. We only provide an informal description with a figure to illustrate CC-SWN model of multiple publishers/subscribers event channels. Moreover, in the figure, we model for clarity only one transition (*Task1*) for the local processing of the subscriber.

Mapping rule 5: Event interfaces (2) The C-SWN of an event channel with multiple publishers is modified (figure 9) by duplicating the *TRE* transition with its corresponding arcs and places (*Rec_Events, CAck*), as many times as there are publishers. If there are multiple subscribers, places *SE_channel, Ack_subscribe* are duplicated (with corresponding arcs), as many times as there are subscribers.

3.4 modeling primitive components

In this section, we derive an SWN model (its *C-SWN*), for each primitive component. Clearly, it is possible to abstract at this stage, the model of a primitive component by choosing an appropriate level of details:

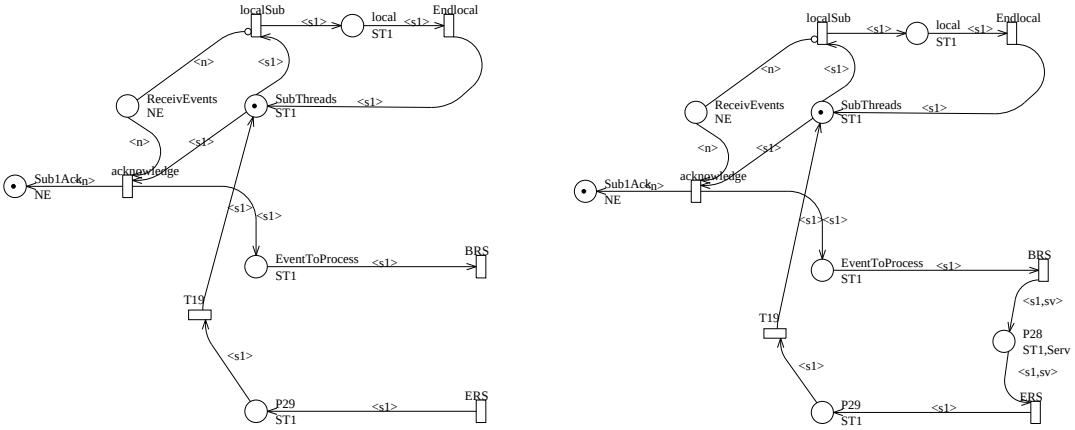
- At the highest abstraction level, the content of a primitive component can be modeled by a very simple SWN.
  This is done when the interest of the modeller doesn't focus on the details of the component, but rather aims at estimating performances at an architectural level of its application.
- At a finer level of details, we model all internal activities of the component.

The C-SWN of a primitive component is then built as follows.

```
PRIMITIVE COMPONENT CC-SWN BUILDING ALGORITHM
BEGIN
1. Analyze the source code of the component, fix a level of details
for modeling.
2. For each set of methods offered by a server interface, model this
interface using mapping rule 1. Model internal activities if required
by the selected level of details.
3. For each service invocation, model the client interface using
mapping rule 1.
4. Complete eventually client and service interfaces models using
mapping rules 2 and 3.
5. Translate into C-SWNs each event and source using mapping rule 4,
providing the event channel model. Model local activities of
subscribers and event channels, depending on the required level of
details.
6. Complete the obtained model for each event channel using mapping
rule 5, whenever several publishers and/or subscribers are involved.
7. If another activity is called within an interface (ie. when a method
of the interface is invoked), model this activity.
END
```

After this modeling step, the obtained CC-SWN is completed with stochastic parameters. This is done by associating rates to transitions of the obtained model. These rates may be estimated through a *model parameters estimation phase*, where a test application (see for instance the *Grinder* tool) is ran in order to measure the parameters needed for performance prediction.

We illustrate the construction of a CC-SWN through modeling of the first StockBroker component. We model first the event sink interface by applying



(a) C-SWN model

(b) CC-SWN model

Figure 10: C-SWN and CC-SWN of the StockBroker 1 component

mapping rule 4. The *ST1* colour class models the threads of the component, whereas the *NE* class represents possible received events. We then build the model of the client interface using mapping rule 1. We obtain the C-SWN of the figure 10(a). Then, we complete the client interface using mapping rule 2. The final CC-SWN is given by the figure 10(b).

3.5 modeling composites

After modeling primitive components of a CBS architecture, we model composites of higher levels. A composite component of a level i is made up of a set of interconnected sub-components of level $i-1$, being themselves primitive or composite. Building the SWN model of a composite requires connecting sub-components CC-SWNs and modeling its external interfaces. This procedure is repeated for each level until reaching the application level. For this purpose, we apply the following algorithm:

```
COMPOSITE CC-SWN BUILDING ALGORITHM
BEGIN
Let N be the number of levels of the composite.
1. Model primitive components of level 0.
2. For (i=1; i<N; i++) do
     a. For each composite C of level i do
          (i) Assemble components of level i-1 :
              For each couple of subcomponents connected by a service
              invocation, fuse corresponding transitions (TBRS,TBPS) and
              (TERS,TEPS).
              For each couple of components connected by an event
              interaction, fuse places of the publisher and event channel
              (SentEvents, Rec_Events) and (Ack_channel,CAck), and places
              of event channel and the subscriber (SE_channel,ReceivEvents)
```

```
            and (Ack_subscriber,SAck).
          (ii) Each not connected interface of a subcomponent is
               considered as an external interface of C.
        EndFor
     b. Model primitive components of level i.
   EndFor
END
```

Fusion of two transitions (resp. places) consists of defining a unique transition (resp. place), and keeping associated arcs of fused transitions (resp. places). Colour classes of the two transitions are mapped in one to one correspondence for common parameters of the interface and specific colour classes of each transition (resp. place) are kept. Hence, the colour domain of the fused transition is the Cartesian product of colour classes of the fused transition (without repetition). Whereas, colour classes of two fused places are mapped in one to one correspondence leading to the colour domain of the fused place.

When assembling SWN models of sub-components, name conflicts (of places, transitions or colour classes) may arise. They are eliminated by renaming. This renaming, as colour classes instantiation, requires at the end of analysis, to restate analysis results (obtained properties and computed performance indices) in the initial context of the CBS. Note that, together with the CC-SWN of the composite, we keep track of the CC-SWNs of its sub-components since they will be used during the analysis phase.

3.6 Modeling container services

A container is made up of a set of interconnected CCM components and a set of services offered to its components. It mediates invocations of components from or to external components belonging to other containers, through: (i) either a *callback* (external) interface which acts as an interceptor for all incoming calls to the component, (ii) or an interceptor for outgoing calls, internal to the container.

So, building the SWN of a container requires connecting its components CC-SWNs, modeling the services and modeling the mediation role.

3.6.1 Connecting the components CC-SWNs

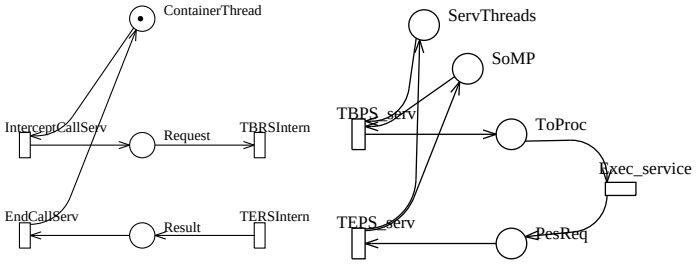
Interconnection of the CC-SWNs is translated into fusion of transition/places corresponding to interfaces of communicating components, as explained in section 3.5.

3.6.2 Modeling container services

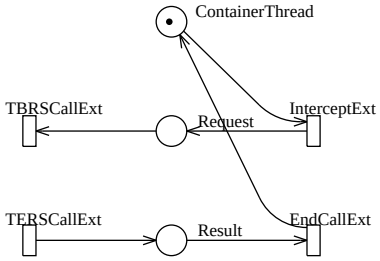
Achieving this modeling requires to associate a CC-SWN model to each container service and a second SWN modeling routing of a component request to the service needed. We assume a monothreaded container. Mapping rule 6 describes this concern illustrated in figure 11.

Mapping rule 6: Container services

• A container service is modeled with an abstracted component having one service interface, identified by a set of server threads colours and offering a set MP of methods (figure 11(b)). One transition *Exec_service* abstracts the service.



(a) Routing service request (b) Service



(c) Callback

Figure 11: Modeling elements related to container services

• Routing a request to an invoked container service is modeled with the model of figure 11(a). A single place *ContainerThread* models the unique thread of the container. The *InterceptCallServ* transition expresses intercepting a request. It is controlled by the *ContainerThread* place. The *TBRSIntern* transition models the request made by the container to its service. The result is obtained with the *TERSIntern* transition and sent to the requester using *EndCallServ* transition.

We choose to abstract the activity induced by a container service, as our goal is to consider the impact of a service on the execution of the analyzed CBS. In another side, the container can manage its component instances, threads or resources using the pooling technique for reducing some overhead. If the designer is interested in knowing the impact of this pooling on performances of his application, we can consider this by associating a consequent rate to the transition *TBRSIntern* related to the invoked operation. Note that, for more clarity, the SWNs given in figure 11 are not coloured, but should be.

3.6.3 Modeling the mediation role

Outgoing component calls are mediated by the container. This is modeled, as in the case of a container service invocation, using mapping rule 6. External invocations to components services located inside a container are also intercepted by this container

on an external interface, and routed to the concerned component. This is modeled with mapping rule 7.

Mapping rule 7: Callback interfaces
Routing an external invocation to a component service of a container is modeled with the model of figure 11(c). The *InterceptExt* transition models the interception of a request. It is controlled by the *ContainerThread* place. The *TBRSCallExt* transition represents the submission of a request to the concerned component. The result is obtained with the *TERSCallExt* transition and sent to the requester using *EndCallExt* transition.

3.6.4 CC-SWN of a container

Modeling a container leads to a CC-SWN interfaces of which are defined as callback interfaces and non-connected interfaces associated with internal invocation of external services.

```
CONTAINER CC-SWN BUILDING ALGORITHM
BEGIN
1. Model each container service using mapping rule 6.
2. Connect communicating CCM components.
3. For each invocation of a container service, build a mediation
   part using mapping rule 6, connect it to the requested component
   on one side, and to the service model on the other side.
4. Model each callback interface using mapping rule 7, and connect
   it to the requested component service.
5. For each internal invocation of a service offered by an external
   component, build a mediation part following mapping rule 6,
   and connect it to the requester component.
END
```

We illustrate the modeling with the SWN given in the middle of figure 12. In this figure, two components $cp1$ and $cp2$ request respectively service1 and service2 of the container. An external client also requests a service from a component internal through the container.

3.7 Building the G-SWN of a CBS

Modeling a CBS application requires interconnection of the CC-SWNs models of all its elements (components, event channels, containers, ...) obtained in previous steps. This is done via connection of communicating interfaces. The obtained model is then completed by "closing" interfaces of the application with a small Petri net to ensure a *finite* state space of the G-SWN . This is a classical method, allowing us to limit the number of entities in the model. In the Petri net context, we add a small Petri net to each interface of the application with an adapted initial marking, generally an upper bound of the number of entities. An example of such a closing SWN is given in figure 13.

For our application example presented in figure 3, we first build CC-SWN models of the StockDistributor components 1 and 2 (figures 14 and 15), the StockBroker components 1 and 2 (figures 10(b) and 16) and the Executor component (figure 17(a)).

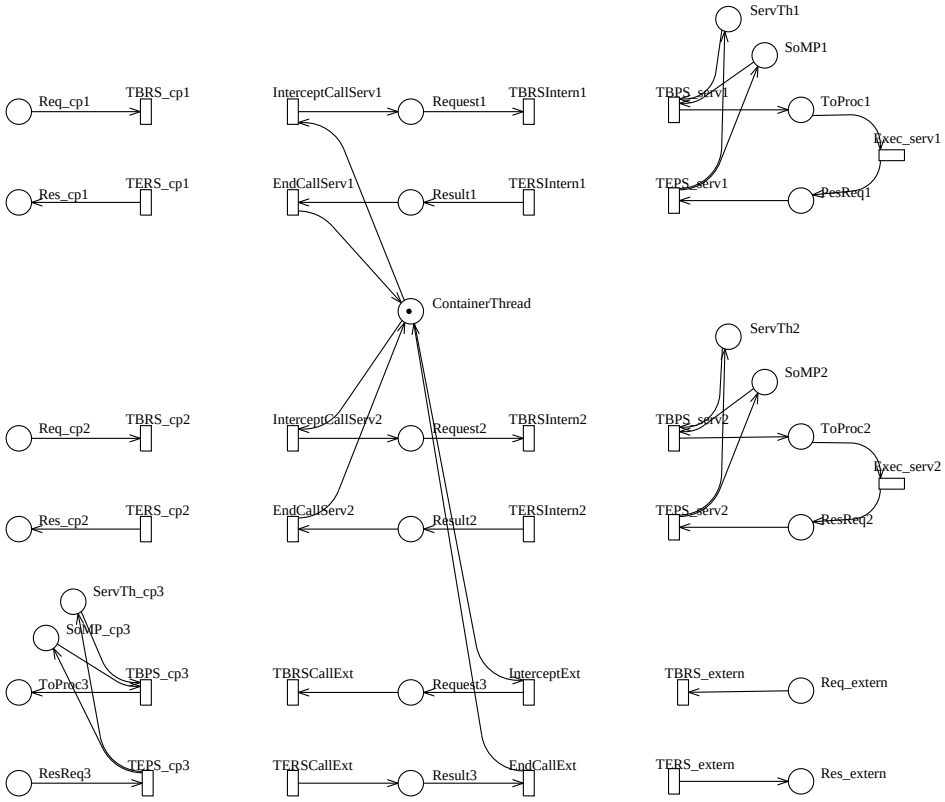


Figure 12: Example of a container CC-SWN

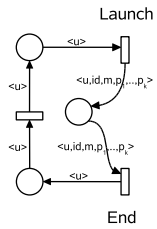


Figure 13: Application closing subnet

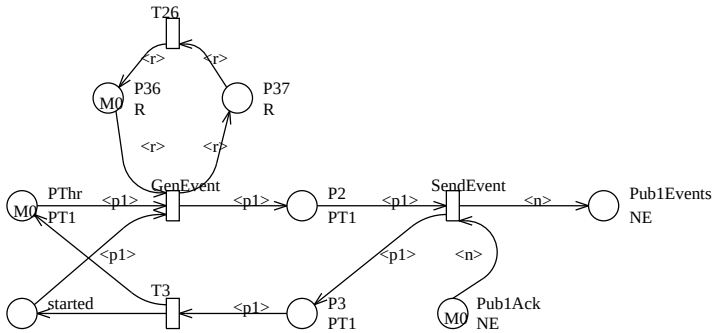


Figure 14: CC-SWN model of the StockDistributor 1

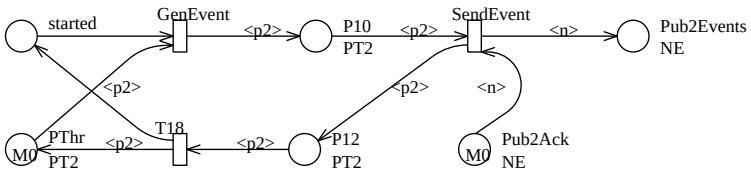


Figure 15: CC-SWN model of the StockDistributor 2

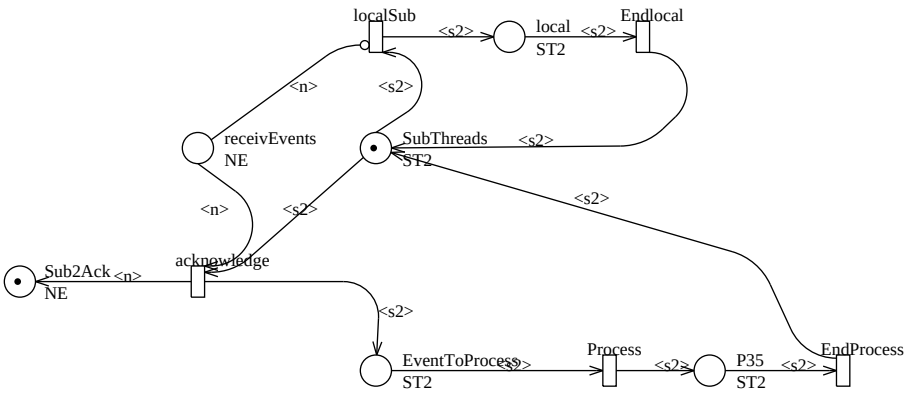
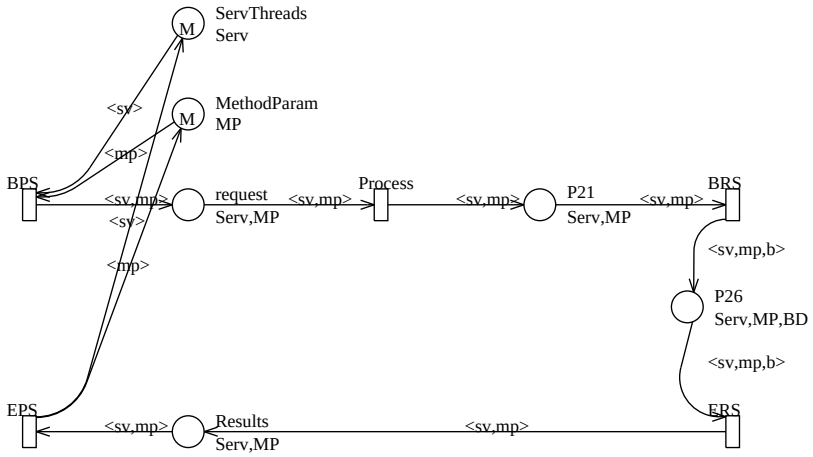
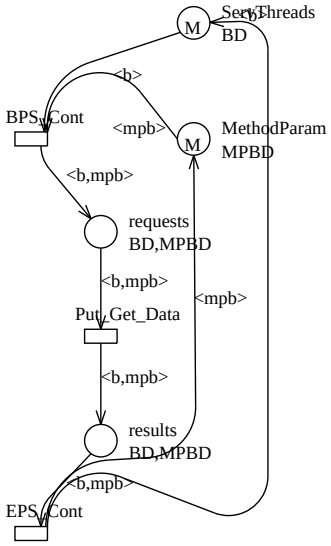


Figure 16: CC-SWN model of the StockBroker 2



(a) CC-SWN model of the Executor



(b) CC-SWN model of the persistence service

Figure 17: CC-SWN models of the Executor and the persistence service

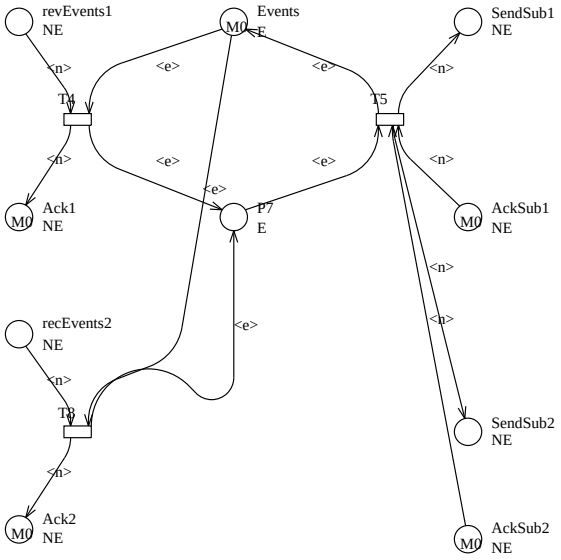


Figure 18: CC-SWN model of the event channel of the stock quoter example

We also build the CC-SWN of the event channel following mapping rule 4. The event channel manages a set E of events for two publishers and two subscribers. So, we apply mapping rule 5 to build the CC-SWN of the channel (figure 18).

On the other hand, the executor component receives requests from the first subscriber on its server interface. Two basic colour classes *Serv* and *MP*, modeling respectively server threads and (method, parameters) pairs, are involved in processing requests.

Storing data is achieved by requesting the container persistence service through a client interface (transitions BRS and ERS) of the executor component. We obtain so the C-SWN which is completed also using mapping rule 2 (figure 17(a)). The persistence service is modeled as an abstracted component given by figure 17(b)). The colour classes BD and $MPBD$ are respectively representing threads of the service and its methods with associated parameters.

Then, we modeled the container which routes request of the Executor component to the persistence service. Finally, we connect all obtained CC-SWNs (of components, event channel, container and persistence service) and close the global model with a "closing" Petri net. We obtain thus the G-SWN of figure 19.

Note that some places and transitions have been renamed because of name conflicts between the two StockDistributors, between the two StockBrokers and between the Executor and the persistence service.

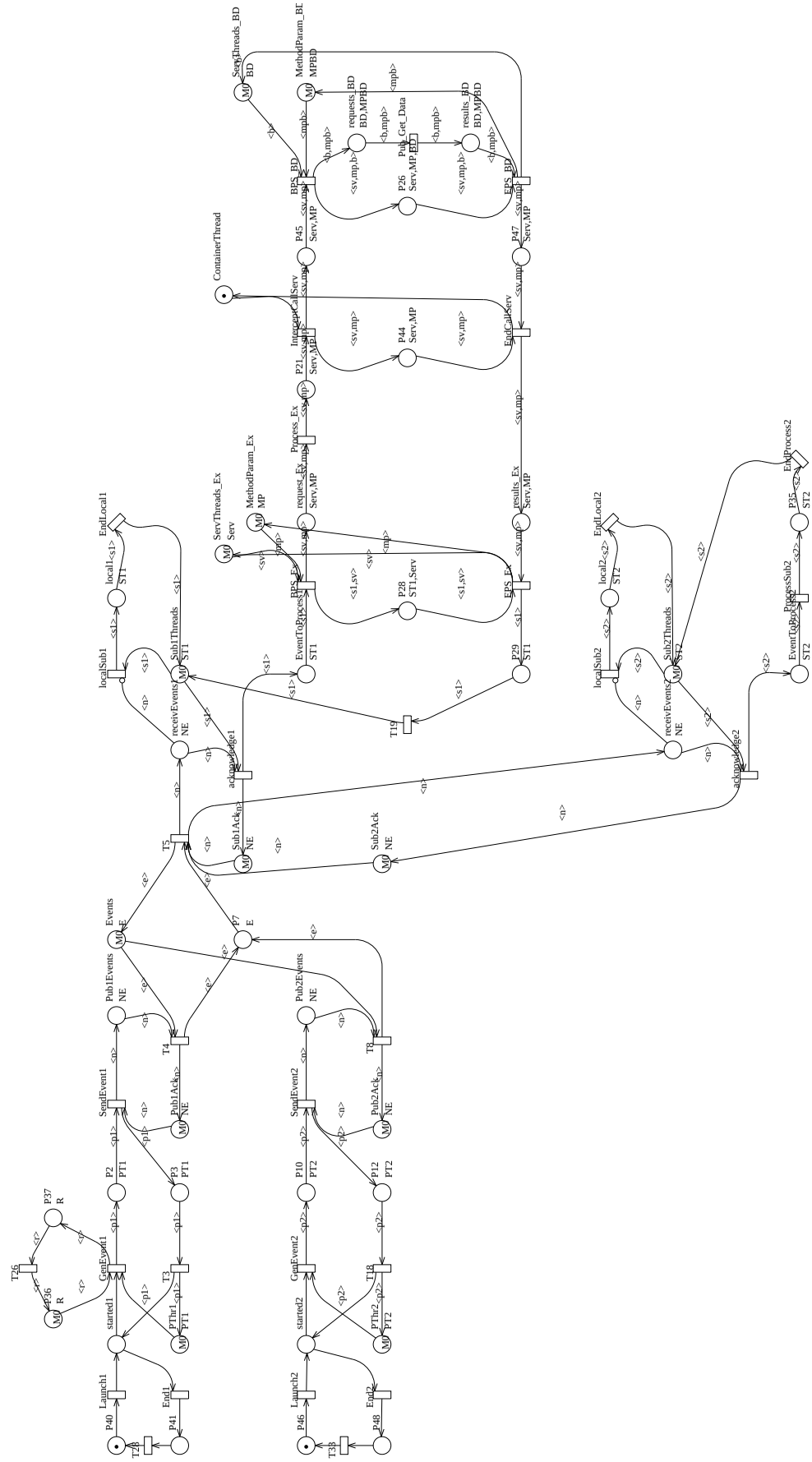


Figure 19: G-SWN model of the stock quoter example

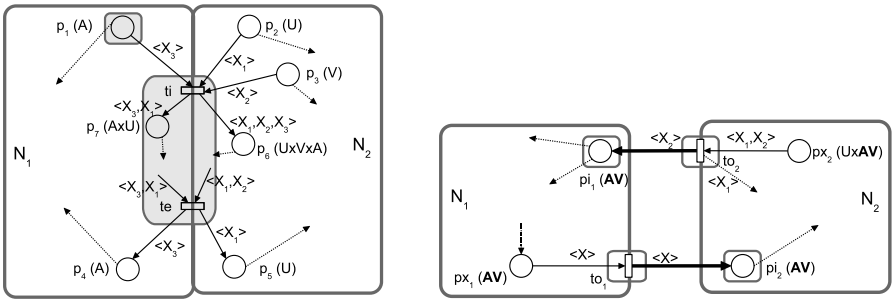


Figure 20: Synchronous (left) and asynchronous (right) decomposition of SWNs

4 Structured performance analysis of CBS

After generating the G-SWN of a CBS and the $(\text{CC-SWN}_k)_{k\in K}$ corresponding to its components, we apply the two last steps of our analysis approach, aiming mainly at computing performance indices of a system such as the response time to a request, the throughput, the mean number of a certain resource, etc. We may also check qualitative properties like deadlocks, reachability of a particular state.

Analysis of a CBS can be performed through the direct analysis of the G-SWN obtained in the first step of our method. This approach has been followed in [3] for analysis of a composition of SWNs, and implemented in the Algebra tool of the GreatSPN package [21]. In our approach, we rather benefit from the compositional structure of CBSs in order to provide an efficient steady-state performance analysis with regard to computation time and memory requirements.

For this purpose, we devise an extension of our previous work [8, 9] adapted to CBS. Let us remind that this approach defines a structured analysis method for a decomposition of SWNs, allowing to avoid explicit construction of the aggregated Markov chain corresponding to the global SWN. The main idea is to decompose a (global) SWN into several subnets, and study each subnet augmented with "parts" abstracting interactions with other subnets. These separated studies are then used to derive a tensorial representation of the generator of the underlying aggregated Markov chain of the global net. The tensorial representation is finally used to compute performance indices which leads to memory and computation time savings.

For this purpose, two kinds of decompositions were defined: (i) a "synchronous" decomposition (figure 20, left) modelling a complex "Rendez-vous" like synchronization between two SWNs;
(ii) an "asynchronous" decomposition (figure 20, right) which corresponds to an asynchronous method call or a message sending and receiving between two or more SWNs. Each kind of decomposition requires a set of conditions which can be checked at the SWN definition level. We refer the reader to [12, 13] for a detailed presentation of these conditions.

4.1 Structured analysis method for CBS

The extension of our structured analysis to CBSs rises three problems:

1. Composition of CC-SWNs of components, as we start from the definition of components in the case of a CBS. This is in contrast to the previous method where a global SWN is decomposed into several subnets. Composition of SWNs models of a CBS has been explained above.

2. Bringing an interconnection of components into a synchronous or asynchronous composition of SWNs. We map a request/response interaction into a synchronous composition of CC-SWNs, while we model an event interaction with an asynchronous composition of CC-SWNs. We emphasize here that our modelling of interfaces *ensures* satisfaction of structural conditions stated for synchronous and asynchronous compositions of subnets given in [12, 13].

3. Impact of synchronous and asynchronous compositions in the same global model, as the structured method was defined for either a synchronous composition or else asynchronous composition of SWNs. This problem leads us to restrict the application of the structured analysis to the following cases:
(i) If $(\mathcal{N}_1, \mathcal{N}_2)$ and $(\mathcal{N}_1, \mathcal{N}_3)$ (resp. $(\mathcal{N}_2, \mathcal{N}_3)$) are in pairwise client/server relationship, then $(\mathcal{N}_2, \mathcal{N}_3)$ (resp. $(\mathcal{N}_1, \mathcal{N}_3)$) are not in client/server relationship.
(ii) If $(\mathcal{N}_1, \mathcal{N}_2)$ are in publish/subscribe relationship, then if they are in client/server relationship too, then event colours are not involved in the client/server interaction.
(iii) If $(\mathcal{N}_1, \mathcal{N}_2)$ and $(\mathcal{N}_1, \mathcal{N}_3)$ are in publish/subscribe relationship, then if $(\mathcal{N}_2, \mathcal{N}_3)$ are in client/server relationship, then event colours are not involved in the $(\mathcal{N}_2, \mathcal{N}_3)$ interaction.

We give below our analysis algorithm based on the structured method. We start with the G-SWN of the application and the CC-SWNs corresponding to the CBS elements (components, channels, containers, ...). Let us denote by E the set $\{\text{CC-}SWN_k \mid 1 \leq k \leq K\}$.
1. Find the set of SWN subnets $(\mathcal{N}_k)_{1 \leq k \leq K'}$ representing a possible decomposition of the G-SWN, that fulfill conditions stated in [12, 13] for a structured representation of the SRG and its aggregated generator. These SWNs do not necessarily correspond to the CC-SWNs of the set E due to restricted conditions above. This point is investigated by checking first service invocation interactions, and then event interactions.
2. Extension of the SWNs $\mathcal{N}_k$ to autonomous SWNs $\bar{\mathcal{N}_k}$. These autonomous SWNs are called *extended nets* (see [12, 13] for details).
3. Generation of the SRGs of these extended SWNs.
4. Computation of the synchronized product of these SRGs and of the tensorial representation of the generator of the underlying aggregated Markov chain.
5. Computation of the steady state distribution of the aggregated model and computation of the required performance indices.
6. Expression of the results in the initial context of the components.

Automation of points 1 and 6 are currently under development, whereas the the others steps have been automated in a tool *compSWN*, the new version of the TenSWN tool [8].

4.2 Illustration

We note that the set of (CC-SWN$_k$) obtained when modelling components satisfy conditions for a structured analysis. We then use our tool *compSWN* on this set of subnets to compute steady-state probabilities.

Cf	—E—	—R—	—Serv—	—MP—	—BD—	—MPBD—
1	1	4	3	5	4	3
2	4	4	3	5	4	3
3	8	4	3	5	4	3
4	12	4	3	5	4	3

Cf	NbS	NbO	TG	TC	MG	MC
1	51840	7538688	147	25	13.46	0.19
2	129600	60309504	537	116	37.13	0.45
3	233280	964952064	1130	382	68.65	0.78
4	336960	15439233024	1650	855	100.17	1.13

Table 1: State space size and time computation for various configurations of the application

Transition	Rate	Transition	Rate
GenEvents1	0.75	BPS	0.9
GenEvents2	0.65	localSub1	8
SendEvents1	0.8	localSub2	8
SendEvents2	0.7	process	0.9

Table 2: Transition rates of the studied configuration

4.2.1 Savings with the structured method

We first show time and memory savings due to the use of the structured analysis. To this end, we use the GreatSPN environment on the G-SWN to compare results of both analysis methods. We vary the cardinalities of our basic colour classes, and we study the behaviour of the solvers for several configurations. The solvers run on a Suse linux 9.2 workstation with 512 MO.

We report in table 1 behaviours of the two solvers (GreatSPN and *compSWN*) for what concerns memory usage (in bytes) and computation times *for the SRG generation phase only* of the resolution, i.e. without computing steady-state probabilities nor performance indices. We also indicate the state space sizes of the global net. Notations for table 1 are the following: —Colour— is the cardinality of the static colour subclass denoted by *Colour*, NbS is the number of symbolic markings, NbO is the number of ordinary markings, TG is the computation time of GreatSPN, TC is the computation time of *compSWN*, MG is the memory (in Megabytes) used by GreatSPN and MC is the memory used by *compSWN*. Note that we assume a cardinality of value 1 for basic colour classes PT1, PT2, ST1 and ST2 of respectively publishers and subscribers, as we considered monothreaded components.

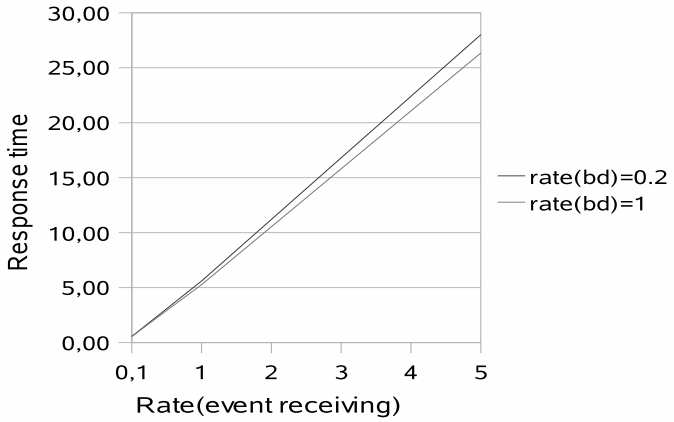


Figure 21: Event processing response time versus event receiving rate for two container service rates

4.2.2 Response time variations

We are interested in studying the variation of three performance indices:
• Response time to process an event in the first subscriber, with respect to the event receiving rate, since it invokes a service from another component, which itself invokes a container service.
• Response time for local processing of the subscriber, with respect to the event receiving rate.
• Response time of a subscriber request processed by the executor component, regarding the event sending rate.

The goal of studying these variations is to evaluate the impact of the container service (for the first indice), and the impact of sending and receiving events on the subscriber activity (local and event processing). For that purpose, we choose configuration 4 (table 1), since the number of symbolic markings is significantly higher than for configurations 1 to 3.

Note that even though we use small cardinalities for colour classes, a colour may model a group of elementary entities, for instance a request colour can stand for 10, 100 or 1000 requests; obviously, firing rates of transitions involving this colour should be adapted to the semantics of a coloured token (100 requests provide a possibly 100 times (or more) slower processing rate for instance).

We take fixed rate values for a critical set of transitions (see table 2) and we vary transition rates (transitions not appearing in this table have rate equal to 1, i.e. faster than all other transitions, rates being given in the same unit):

• For the first variation, we fix the rate of the *BPersistServ* transition of the persistence service model to a value (0.2 then 1); we vary the rate of the *receiveEvent1* transition of the first subscriber, then we compute the response time for processing an event. We obtain the diagram of figure 21.
• For the second and third variations, we take several values for the *SendEvent1*

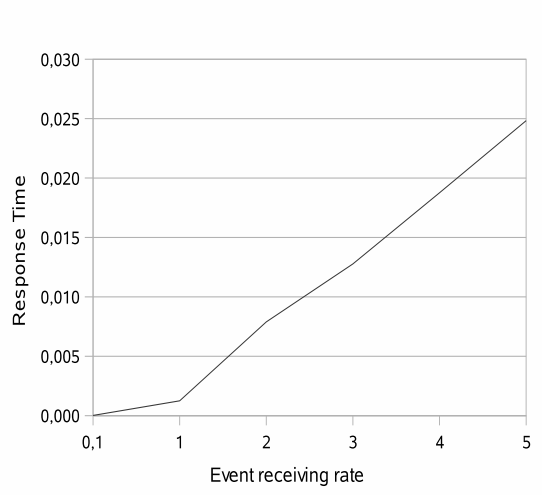


Figure 22: Subscriber local activity response time versus event sending rate

and respectively *receiveEvent1* transitions and we compute the response time for a subscriber request to the executor component and for a local activity of the subscriber. (figure 22).

The first diagram shows an increasing response time as the receiving rate increases for the two curves. However, we can see an improved (reduced) response time in the second curve, corresponding to a higher processing rate of the persistence service. This phenomenon proves that container services have an important impact on the performance of the whole system. In the second diagram, the response time is slowly increased in the first period of time, and gets rapidly augmented as the event receiving rate increases. This is expected, since event processing slows down local processing. Last computations show a very neglecting impact of the event sending on the processing of a subscriber request, as the obtained response time remains approximately the same (61.1 for event sending rates ranging from 0.1 to 5). That is an expected result since events do not interrupt the executor component.

5 Conclusion

Throughout this paper, we have proposed an approach allowing to study, in an efficient way, performances of Component Based Systems. The approach starts from the description of a CB designed application, given in an Architecture Description Language. We translate each component into an SWN model describing its functional behaviour and, partially, its non functional behaviour, using systematic mapping rules. Then, we connect the SWNs of the components following the architecture of the system, starting from primitive components to highest level composite components.

The main advantage of our proposed modelling for CBS is its suitability for applying our adapted structured analysis of SWNs composition, since modelling ensures required conditions for pairewise interactions of components. Our

structured analysis, based on a tensorial representation of the symbolic reachability (SRG) of the G-SWN of the studied CBS, allows us to avoid the explicit construction of the aggregated Markov chain corresponding to the G-SWN. Hence, it enables important memory and computation time savings.

We instantiated this method to Fractal and CCM CBS in our papers [27, 26]. We are currently developing automation of the various steps of the method: extraction of information from the ADL descriptions of the CBS and combination of models of sub-components to enable structured analysis.

References

[1] T. Barros, A. Cansado, E. Madelaine, and M. Rivera. Model checking distributed components: The Vercors platform. In *3rd workshop on FACS*. ENTCS, Sep 2006.

[2] T. Barros, L. Henrio, and E. Madelaine. Behavioural models for hierarchical components. In *Model Checking Software, 12th International SPIN Workshop*, volume LNCS 3639, pages 154–168, San Francisco, CA, USA, August 2005. Springer.

[3] S. Bernardi, S. Donatelli, and A. Horváth. Implementing compositionality for stochastic Petri nets. *Int. J. STTT*, 3(4):417–430, 2001.

[4] A. Bertolino and R. Mirandola. Software performance engineering of component-based systems. In *Proc. of the WOSP 04*, Redwood City, C.A., January 14–16 2004.

[5] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *Proc. of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP02)*, June 2002.

[6] E. Bruneton, T. Coupaye, and J.B. Stefani. The fractal component model, version 2.0-3. Technical report, Fractal team, `http://fractal.objectweb.org/specification/` (Oct. 2006), Feb 2004.

[7] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic well-formed colored nets and symmetric modeling applications. *IEEE Trans. on Comp.*, 42(11):1343–1360, Nov 1993.

[8] C. Delamare, Y. Gardan, and P. Moreaux. Efficient implementation for performance evaluation of synchronous decomposition of high level stochastic Petri nets. In *Proc. of the ICALP2003*, pages 164–183, Eindhoven, Holland, June 21-22 2003. University of Dortmund, Germany.

[9] C. Delamare, Y. Gardan, and P. Moreaux. Performance evaluation with asynchronously decomposable SWN: implementation and case study. In *Proc. of the 10th Int. Workshop on PNPM03*, pages 20–29, Urbana-Champaign, IL, USA, September 2–5 2003. IEEE Comp. Soc. Press.

[10] L. Dias da Silva and A. Perkusich. Composition of software artifacts modelled using colored Petri nets. *Science of Computer Programming*, 56(1-2):171–189, Apr 2005.

[11] V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *J. Syst. Softw.*, 80(4):528–558, 2007.

[12] S. Haddad and P. Moreaux. Aggregation and decomposition for performance evaluation of synchronous product of high level Petri nets. Document du Lamsade 96, LAMSADE, Université Paris Dauphine, Paris, France, September 1996.

[13] S. Haddad and P. Moreaux. Asynchronous composition of high level Petri nets : a quantitative approach. In *Proc. of the 17th ICATPN*, volume 1091, pages 193–211. LNCS, 1996.

[14] J.M. Ilié, S. Baarir, M. Beccuti, C. Delamare, S. Donatelli, C. Dutheillet, G. Franceschinis, R. Gaeta, and P. Moreaux. Extended SWN solvers in GreatSPN. In *Proc. of the 1st Int. Conf. on QEST 2004*, Enschede, The Netherlands, September 27–30 2004. IEEE Comp. Soc. Press.

[15] N. Medvidović and R. N. Taylor. A classification and comparison framework for software architecture description languages. In *IEEE Trans. On Soft. Eng.*, volume 26, pages 70–93, 2000.

[16] P. Müller, C. Stich, and C. Zeidler. Components@work: Component technology for embedded systems. In *Proc. of the Component-based Software Engineering Track at the 27th IEEE Euromicro Conference (Euromicro CBSE'01)*, Sep 2001.

[17] Object Management Group. The common object request broker: Architecture and specification, 3.0.2 edition. Technical report, OMG, Dec 2002.

[18] Object Management Group. Common object request broker architecture (CORBA) - specification, version 3.1, part 2: CORBA interfaces. `http://www.omg.org/cgi-bin/doc?pas/04-08-02.pdf` (July 2007), 2004.

[19] Object Management Group. Notification service. version 1.1. `http://www.omg.org/technology/documents/formal/notification_service.htm` (April 2007), October 2004.

[20] Object Management Group. CORBA component model specification. version 4.0. `http://www.omg.org/cgi-bin/apps/doc?formal/06-04-01.pdf` (April 2007), April 2006.

[21] Perf. Eval. Group. GreatSPN home page: `http://www.di.unito.it/~greatspn`, 2002.

[22] D. Petriu, C. Shousha, and A. Jalnapurkar. Architecture-based performance analysis applied to a telecommunication system. *IEEE Transactions on Software Engineering*, 26(11):1049–1065, 2000.

[23] A.E. Rugina, K. Kanoun, and M. Kaaniche. A system dependability modeling framework using AADL and GSPNs. Technical Report 05666, LAAS, Nov 2006.

[24] N. Salmi, P. Moreaux, and M. Ioualalen. Formal models of fractal component based systems for performance analysis. In *Proc. of ISoLA 2007 Workshop On Leveraging Applications of Formal Methods, Verification and Validation*, pages 49–60, Poitiers, France, 2007.

[25] N. Salmi, P. Moreaux, and M. Ioualalen. Structured analysis of component based systems: an EJB/CORBA middleware application. In *Proc. of the 1st Int. Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'2007)*, Algiers, Algeria, 2007.

[26] N. Salmi, P. Moreaux, and M. Ioualalen. From architectural design to swn models for compositional performance analysis of component based systems: application to ccm based systems. In *Proc. of the 24th UKPEW 2008 Performance Engineering Workshop*, pages 123–136, Imperial College. London, UK, 2008.

[27] N. Salmi, P. Moreaux, and M. Ioualalen. Performance evaluation of fractal component based systems. *Annals of Telecommunications. Special issue : Software component: The Fractal Initiative*, 2008.

[28] D.C. Schmidt and S. Vinoski. Object interconnections: The CORBA component model: Part 2, defining components with the IDL 3.x types. *C/C++ Users Journal*, April 2004.

[29] L. Seinturier and F. Loiret. état de l'art sur les modèles de composants pour le temps réel et l'embarqué. Technical report, ANR-ARA Sécurité, Systèmes embarqués et Intelligence ambiante, `http://reve.futurs.inria.fr/livrables/T0+6/d111.pdf(28/01/2007)`, June 2006.

[30] Connie U. Smith and Lloyd G. Williams. New book - performance solutions: A practical guide to creating responsive, scalable software. In *Int. CMG Conference'01*, pages 355–358, 2001.

[31] Sun Microsystems. J2EE connector architecture. Technical report, Sun Microsystems, Inc., `http://java.sun.com/j2ee/connector`(jan. 2006), 2006.

[32] C. Szyperski. Component technology - what, where, and how? In *Proc. 25th Int. Conf. on Software Engineering*, pages 684–693. IEEE, May 3–10 2003.

[33] E. Weyuker and F. Vokolos. Experience with performance testing of software systems: issues, an approach and case study. *IEEE Transactions on Software Engineering*, 26(12):1147–1156, 2000.